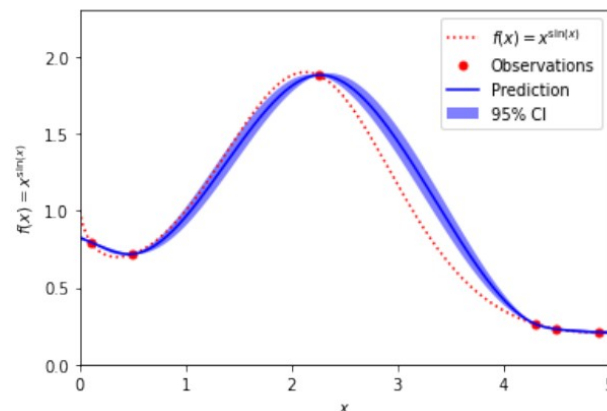# SURROGATE MODELS
FOR STRUCTURAL ESTIMATION AND UNCERTAINTY QUANTIFICATION

University of Geneva
February 28th, 2025

https://github.com/sischei/Deep_Learning_Geneva_2025

Simon Scheidegger
simon.scheidegger@unil.ch

UNIL | Université de Lausanne

Roadmap of this lecture

I. What are are (deep) surrogate models? What are they useful for?

II. A DSGE model with pseudo-states to create surrogates

III. A complement to Deep learning: Gaussian Processes
    I. Basics of Gaussian Process Regression
    II. Noise-free kernels
    III. Kernels with noise
    IV. Bayesian active learning

I. Confronting Models to data

- Contemporary models very rich (many endogenous states, exogenous states, strong non-linearities, lots of parameters,...).

- Expensive to compute.

- Consequently, economists are often forced to sacrifice certain features of the model in order to reduce model dimensionality.
  - estimate only a partial set of parameters while prefixing the others.
  - estimate the model only once using the full sample.

- The high computational costs limit a researcher's ability to carry out a variety of important model analyses.

- Model **estimation**, **calibration**, and **uncertainty quantification** can be daunting numerical tasks
  - → because of the need to perform sometimes hundreds of thousands of model evaluations to obtain converging estimates of the relevant parameters and converging statistics

  (see, e.g., Fernández-Villaverde, Rubio-Ramrez, and Schorfheide, 2016; Fernandez-Villaverde and Guerrn-Quintana, 2020; Iskhakov, Rust, and Schjerning, 2020; Igami, 2020, among others).

A standard solution in science and engineering: Look-up tables

$$\Pr(z \le z_1) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z_1} e^{-\frac{1}{2}z^2} \, dz$$

z_1	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
1.0	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9131	0.9147	0.9162	0.9177
1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
2.0	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817
2.1	0.9821	0.9826	0.9830	0.9834	0.9838	0.9842	0.9846	0.9850	0.9854	0.9857

Surrogate Models: 21$^{\text{st}}$ Century "Lookup Table"
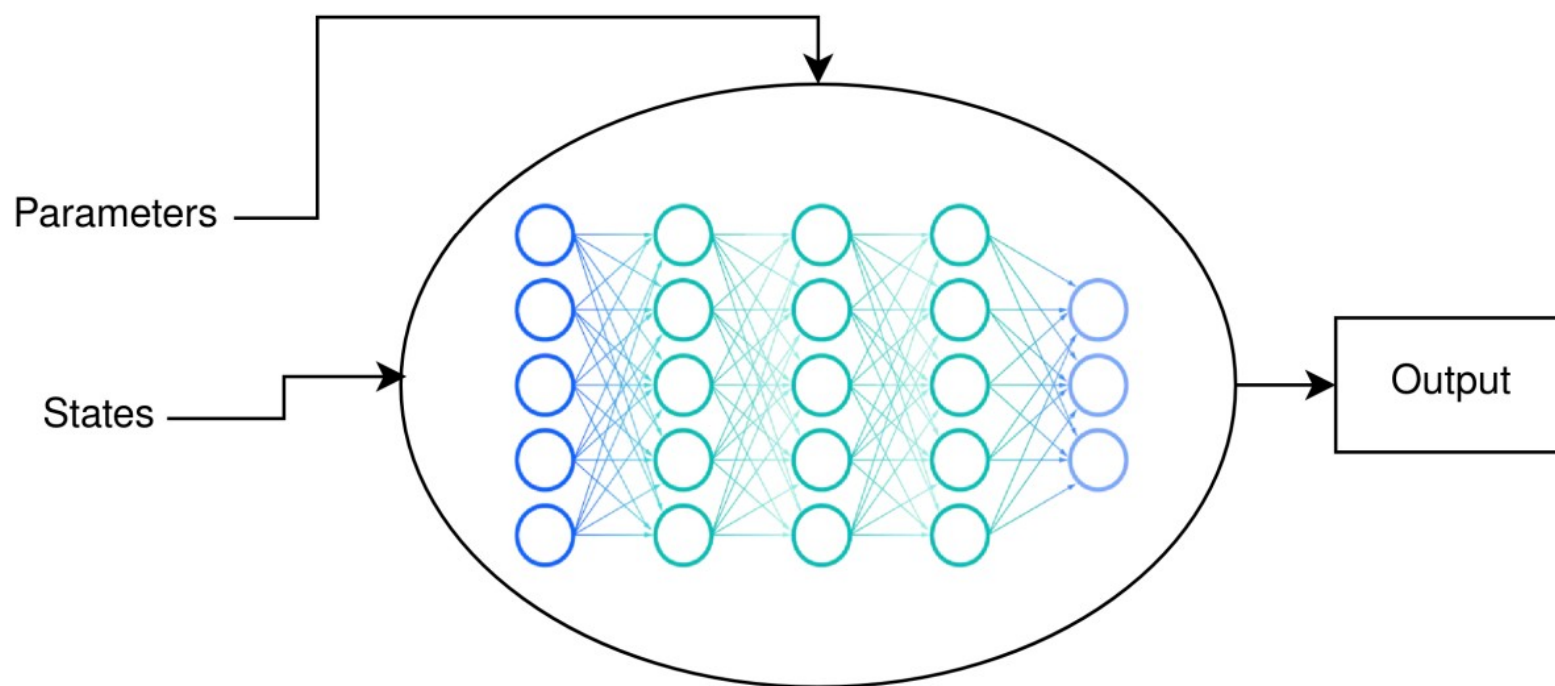
$$\phi(\mathbf{x}_t | \theta_{NN}) = y_t$$

The basic idea

- Replace the economic model with a surrogate!

- Consider a model

$$f : \mathbb{R}^m \to \mathbb{R}^k = f(\Omega_t, H_t | \Theta) = y_t$$

  - where Ω_t is a vector of dimension ω containing the observable states

  - H_t is a vector of dimension h comprising the hidden states

  - Θ is a vector of dimension θ containing model parameters

  - y_t is a vector of dimension k comprising the predicted quantities of interest (such as simulated moments, social cost of carbon in a given year, etc.)

The basic idea (II)

- The problem is that f (·) can be computationally costly, so we wish to construct a cheap to evaluate surrogate, i.e., a Neural Network that replaces the "true" function f (·):
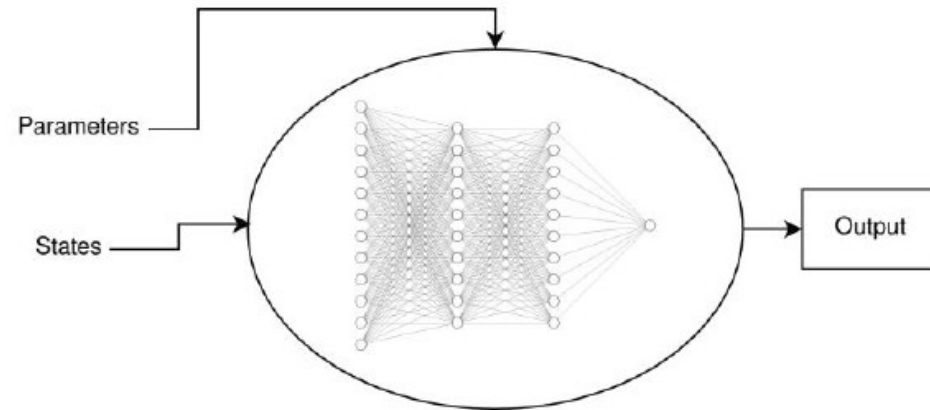
$$\hat{f}(\Omega_t, H_t, \Theta) = \hat{f}(X_t) = y_t$$

- We introduce **parameters as pseudo-state variables** (cf. Norets (2012), Scheidegger & Bilionis (2019))

  $X_t = [\Omega_t , H_t, \Theta]^T$.

- **Solve model only once**, as a function of X_t (global solution) e.g. by using Deep Learning, e.g., by DEQN.

- For reasonable parameter ranges, you may have to use "expert knowledge".

Why (deep) surrogate?
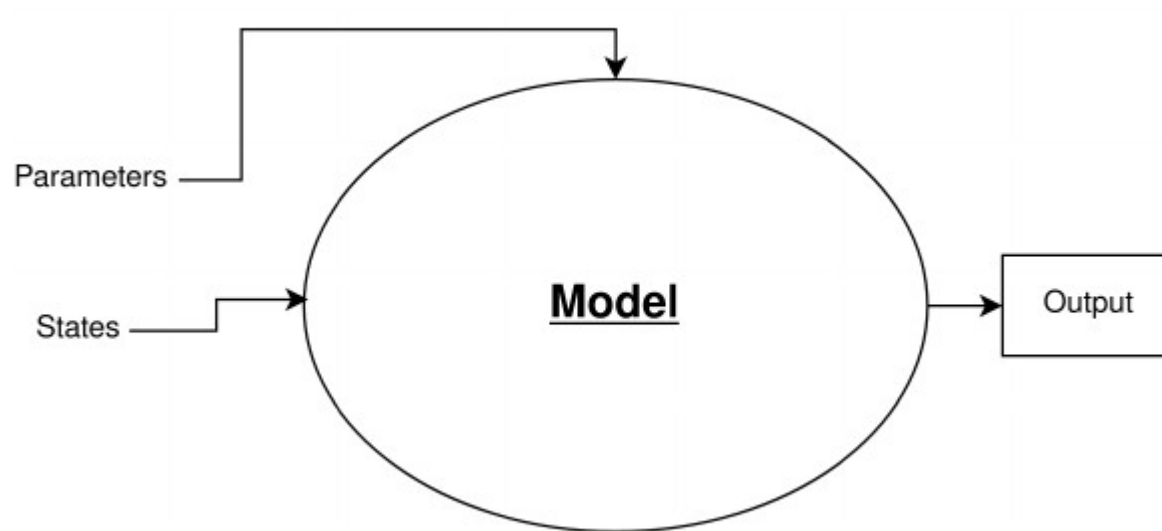
Why (deep) surrogate?

$f(\text{states}, \underbrace{\text{parameters}}_{\text{Pseudo-states}}) = \text{output}$

Some remarks

Deep surrogate is different from standard ML:
- Compared to other methods, deep neural networks are more hungry for data.
- The cost of producing a large training sample should be an important consideration.
- Unlike in standard ML, we know the true model $\Rightarrow$ unlimited data (only limited by
- computational resources); essentially no errors.
- Double descent: Use a large number of epochs
  - $\rightarrow$ Stephenson and Lee (2021); Nakkiran et al., (2021)

Once trained, the deep surrogate
- is highly accurate;
- is cheaper to use by orders of magnitude; makes the gradients readily available;
- is easy to store (for 10^6 parameters - 20 MB vs. $\sim 10^6$GB when using Cartesian grid).

Pay the cost upfront; use it for free later.
- High quality surrogates can be shared with and build on by a community.
- Deep surrogates for workhorse quantitative economic and financial models.

Structural estimation using deep surrogate

- Once we have the deep surrogate, it is easy to use it to for structural estimation.

- For example, suppose we have a set of moment conditions $\mathrm{E}[f(\Omega_t|\theta)] = 0$

- In finite sample, we have $g_n(\theta) = \dfrac{1}{n} \sum_{i=1}^{n} f(\Omega_t, \theta)$

- GMM estimator: $\hat{\theta}_{GMM} = \underset{\theta}{\arg\min}\, g_n'(\theta)\, W\, g_n(\theta)$

- We can use the deep surrogate φ() to replace f (). Moreover, through back-propagation and automatic differentiation the surrogate provides the gradients of φ(·) analytically, which translate the GMM FOCs into a system of nonlinear equations.

How costly is it?

- https://github.com/DeepSurrogate/OptionPricing

- SPX: 4000 option prices on a typical day (Bates option pricing model)

- Horse race:
  - Modern FFT implementation vs. deep surrogate
  - Single core vs. GPU

	FFT	Deep Surrogate	Deep Surrogate + GPU
pricing, 1-day	10s	0.6s	0.06s
estimation, 1-day	180s per iter	3.2s per iter	0.3s per iter
estimation, 1-year	125h	2.2h	.2h

Static data

https://github.com/DeepSurrogate/OptionPricing
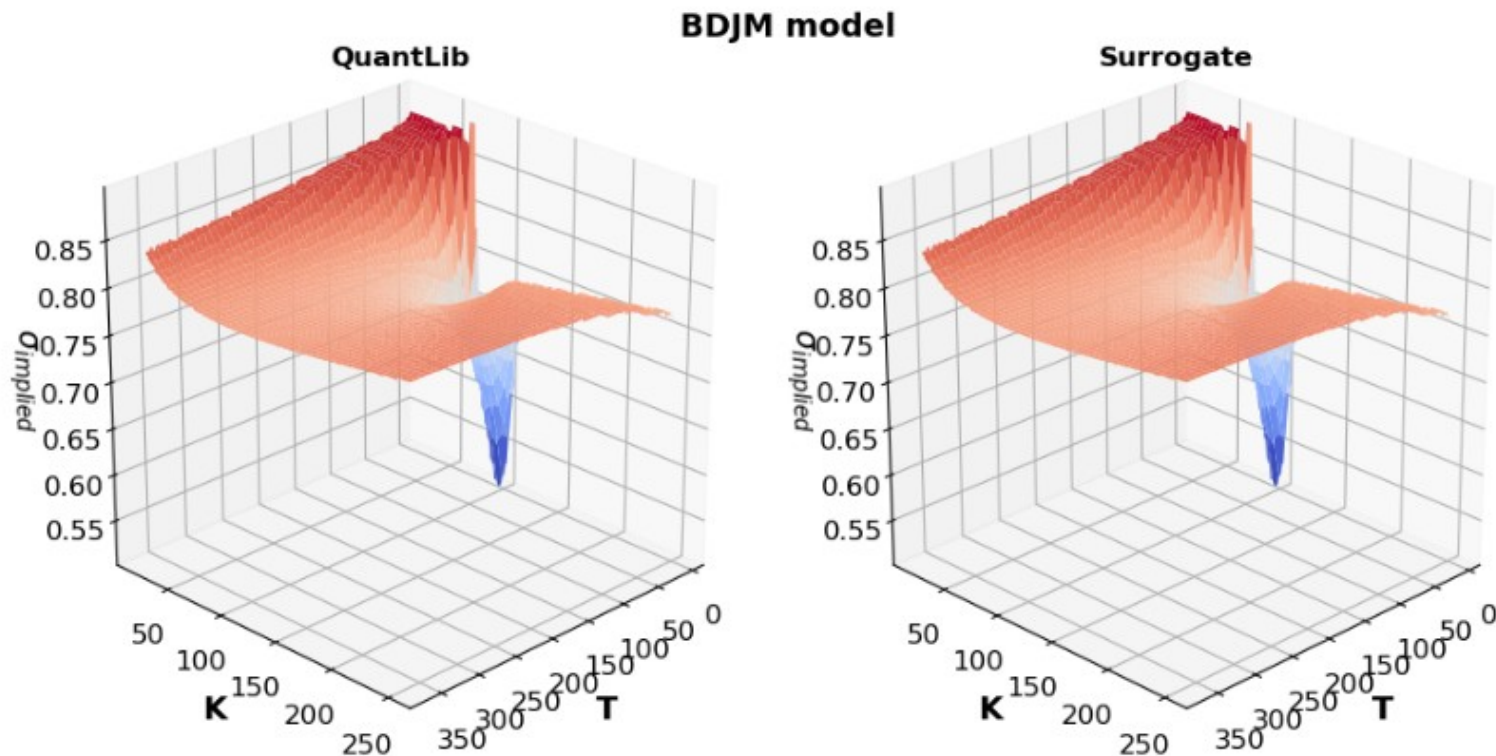
The figure above the implied volatility surface of the Bates model with Double Jump Exponential generated with QuantLib (left) and the Deep-Surrogate (right)

Comparison of approximation methods

	Polynomials	Splines	(Adaptive) sparse grids	Gaussian processes	Deep neural networks
High-dimensional input	✓	✗	✓	✓	✓
Capturing local features	✗	✓	✓	✓	✓
Irregularly-shaped domain	✓	✗	✗	✓	✓
Large amount of data	✓	✓	✓	✗	✓

If data is expensive, GPs are a good alternative

Example: DEQN with pseudo-states

- Let's have a look at a stochastic growth-model with parameters as pseudo-states.

- Code: day2/code/DEQN_production_code/stochastic_growth_pseudostates

  → Solutions can be used to generate to simulate moments, and other quantities of interest.

II. DEQN surrogate

The planner's problem is

$$\max_{C_t, K_{t+1}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t b_t \frac{C_t^{1-\tau} - 1}{1 - \tau}, \tag{1}$$

subject to the resource constraint

$$C_t = A_t K_t^\alpha + (1 - \delta) K_t - K_{t+1} \quad \left(\text{multiplier } \beta^t b_t \mu_t\right) \tag{2}$$

and the irreversability condition

$$K_{t+1} - (1 - \delta) K_t \geq 0 \quad \left(\text{multiplier } \beta^t b_t \lambda_t\right). \tag{3}$$

The Lagrangian takes the form

$$\max_{C_t, K_{t+1}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t b_t \left\{ \frac{C_t^{1-\tau} - 1}{1 - \tau} + \mu_t \left[A_t K_t^\alpha + (1 - \delta) K_t - K_{t+1} - C_t \right] + \lambda_t \left(K_{t+1} - (1 - \delta) K_t \right) \right\}. \tag{4}$$

The Kuhn-Tucker conditions take the form:

$$(C_t) \quad : \quad 0 = C_t^{-\tau} - \mu_t \tag{5}$$

$$(K_{t+1}) \quad : \quad 0 = -\mu_t + \lambda_t + \beta \mathbb{E}_t \left\{ \frac{b_{t+1}}{b_t} \left(\mu_{t+1} \left[\alpha A_{t+1} K_{t+1}^{\alpha-1} + (1 - \delta) \right] - \lambda_{t+1} (1 - \delta) \right) \right\} \tag{6}$$

$$(CS) \quad : \quad 0 = \lambda_t \left[K_{t+1} - (1 - \delta) K_t \right]. \tag{7}$$

Define $d_t = b_t / b_{t-1}$. We assume that the exogenous shock processes evolve according to

$$\ln A_t = (1 - \rho_A) \ln A_* + \rho_a \ln A_{t-1} + \sigma_a \epsilon_{a,t} \tag{8}$$

$$\ln d_t = \rho_d \ln d_{t-1} + \sigma_d \epsilon_{d,t}. \tag{9}$$

III. Gaussian Process Regression

Gaussian Processes for
Machine Learning

Carl Edward Rasmussen and Christopher K. I. Williams

http://www.gaussianprocess.org/gpml/
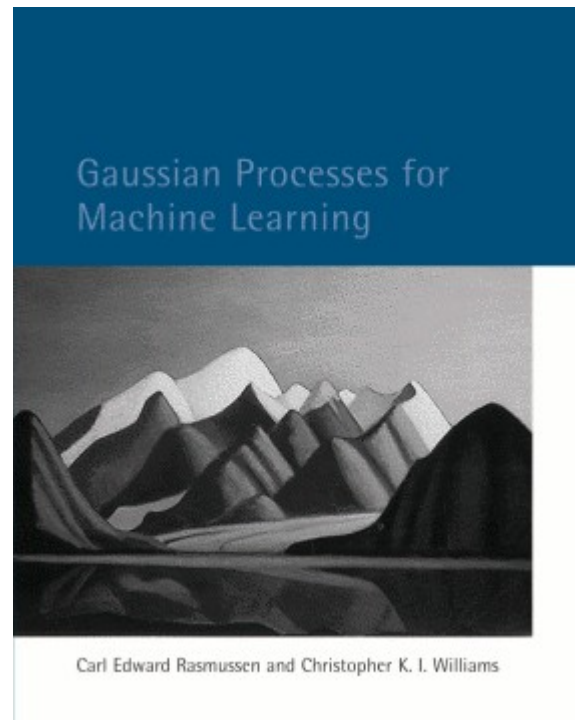
Recall: Aim of Regression

- Given some (potential) noisy observations of a dependent variable at certain values of the independent variable x, what is our best estimate of the dependent variable y at a new value, x $_*$?

- Let f denote an (unknown) function which maps inputs x to outputs

$$f:X \rightarrow Y$$

- Modeling a function f means mathematically representing the relation between inputs and outputs.

- Often times, the shape of the underlying function might be unknown, the function can be hard to evaluate, or other requirements might complicate the process of information acquisition.
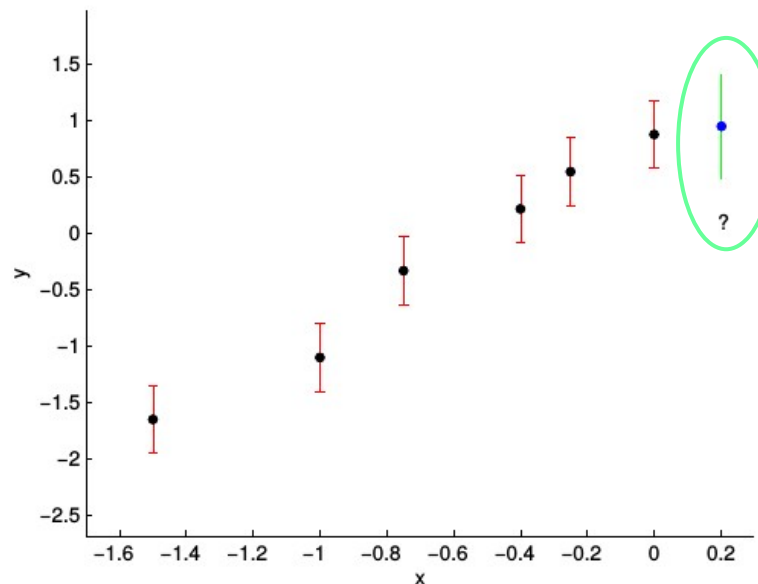
Choosing a model

- If we expect the underlying function f(x) to be linear, and can make some assumptions about the input data, we might use a least-squares method to fit a straight line (linear regression).

- Moreover, if we suspect f(x) may also be quadratic, cubic, or even non-polynomial, we can use the principles of model selection to choose among the various possibilities.

Model Selection

One common approach to reliably assess the quality of a machine learning model and avoid over-fitting is to randomly split the available data into

- training data (~70% of the data)
  is used for determining optimal coefficients.

- validation data (~20% of the data) is used for model selection (e.g., fixing degree of polynomial, selecting a subset of features, etc.)

- test data (~10% of the data) is used to measure the quality that is reported.
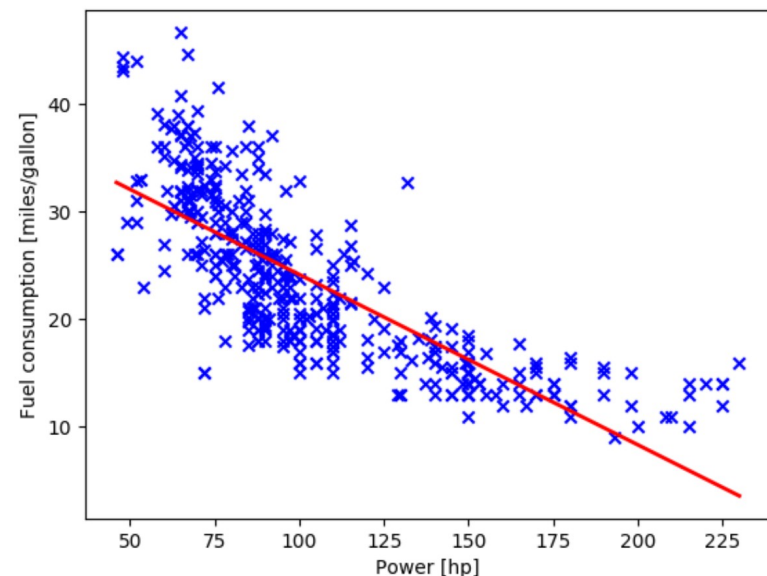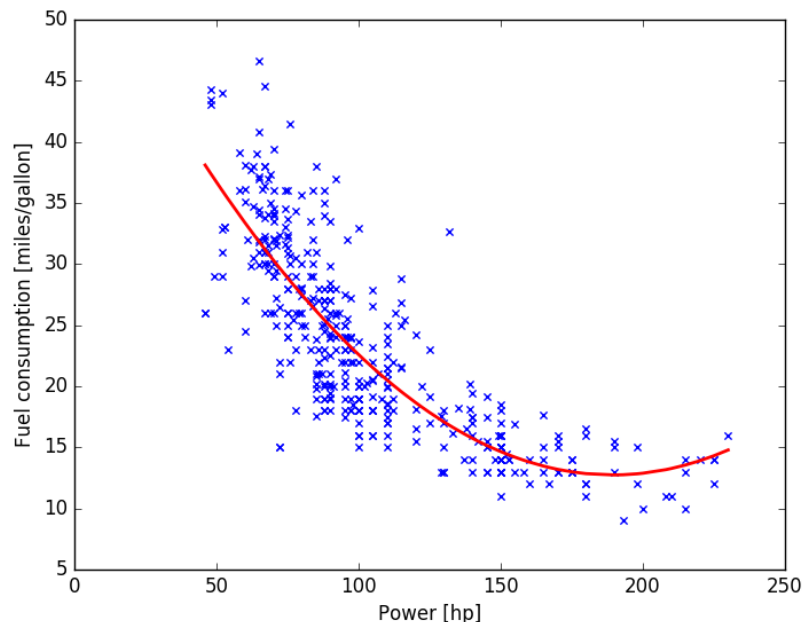
For completeness: Polynomial Regression in Python

See: day2/demo/poly_reg.ipynb

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model, preprocessing

# read cars dataset, a clean data set
cars = pd.read_csv('auto-mpg.data.txt',header=None, sep='\s+')


# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower values
X = cars.iloc[:,[3]].values
X = X.reshape(X.size, 1)

# precompute polynomial features
poly = preprocessing.PolynomialFeatures(2)
Xp = poly.fit_transform(X)

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(Xp,y)

# coefficients
reg.intercept_  # 56.900099702112925
reg.coef_  # [-0.46618963, 0.00123054]

# compute correlation coefficient
np.corrcoef(reg.predict(Xp),y) # 0.82919179 (from 0.77842678)

# compute mean squared error (MSE)
sum((reg.predict(Xp) - y)**2) / len(y) # 18.984768907617223 ( from 23.943662938603104)

### plot

hp = cars.iloc[:,3].values
mpg = cars.iloc[:,0].values

hps = np.array(sorted(hp))
hps = hps.reshape(hps.size, 1)
hpsp = poly.fit_transform(hps)

plt.scatter(hp, mpg, color='blue', marker='x')
plt.plot(hps, reg.predict(hpsp), color='red', lw=2)
plt.xlabel('Power [hp]')
plt.ylabel('Fuel consumption [miles/gallon]')
plt.show()
```

Degree of regression
1: linear
2: quadratic
...

Why Gaussian Process Regression?

- There are many projections possible.

- We have to choose one either a priori or by model comparison with a set of possible projections.

- Especially if the problem is to explore and exploit a completely unknown function, this approach will not be beneficial as there is little guidance to which projections we should try.

Gaussian process regression offers a principled solution to this problem in which projections are chosen implicitly, effectively leading "**the data decide**" on the complexity of the function.

Recall Multivariate Gaussians

Say you measure two variables, e.g.,

- x_1:height
- x_2: weight

$\rightarrow$ plot
$\rightarrow$ we want to fit a Gaussian to these points.

Recall Multivariate Gaussians (II)

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2$$

Mean $\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$
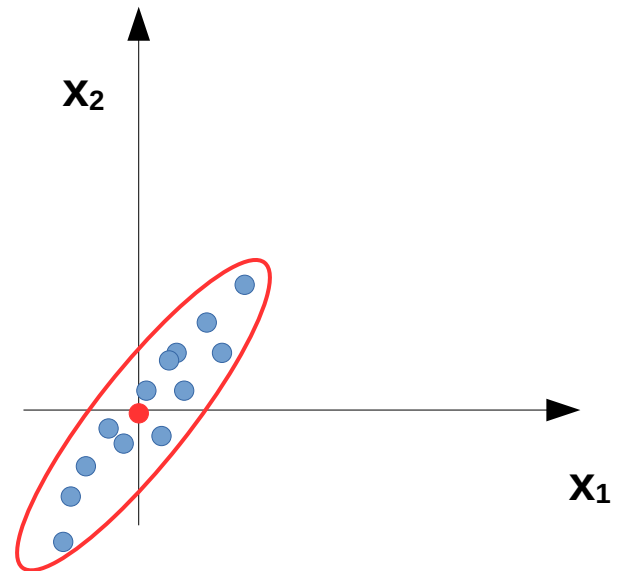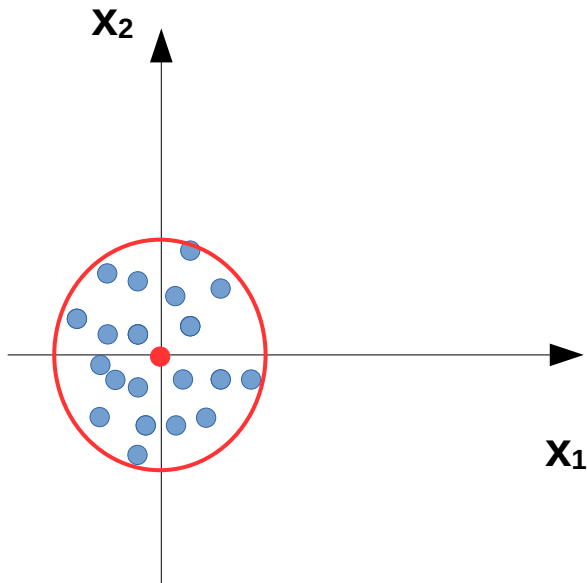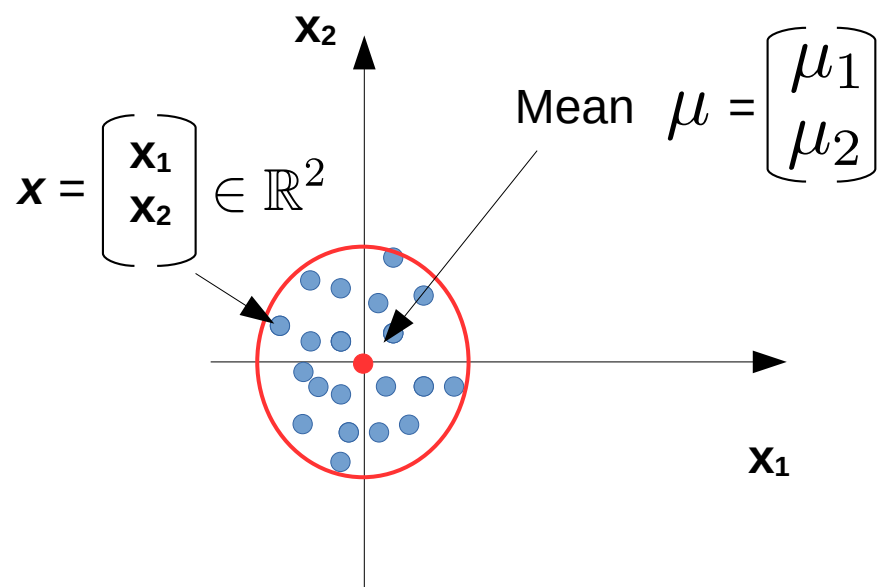
Fit a Gaussian that with a covariance that is circular.

Mean
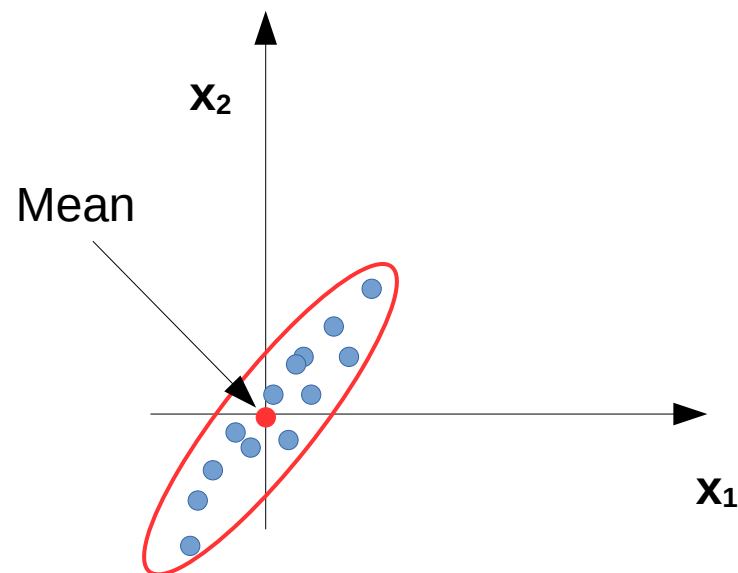
Fit a Gaussian that with a covariance that is an ellipse.

Multivariate Gaussians (III)

- Assume the points are Gaussian distributed (this is our "model").

- How do points relate to each other? ("how does increasing x_1 increase x_2?)

  → The variable to describe this is called "Covariance*" (cov)

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N\left[\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} , \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right]$$

Mean   Covariance

- If the entries in the column vector $\mathbf{X} = (X_1, X_2, \ldots, X_n)^{\mathrm{T}}$
  are random variables, each with finite variance and expected value, then the covariance matrix $\mathbf{K_{XX}}$ is the matrix whose (i ,j) entry is the covariance.

$$K_{X_i X_j} = \mathrm{cov}[X_i, X_j] = \mathrm{E}[(X_i - \mathrm{E}[X_i])(X_j - \mathrm{E}[X_j])]$$

*Covariance matrix: positive definite (Matrix symmetric, its Eigenvalues positive).

Multivariate Gaussian (IV)

- Assume for a moment that **E**[ ] = 0 → Covariance **E**[$x_1 x_2$] is a **"dot" product**.

- Assume two points [1 0] $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ = 1 → **Covariance is a measure similarity**.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)$$

Knowing about **x_1** does not provide any information about **x_2** as they are uncorrelated.

E[$x_1 x_2$] = 0

x$_2$ Mean

x$_1$
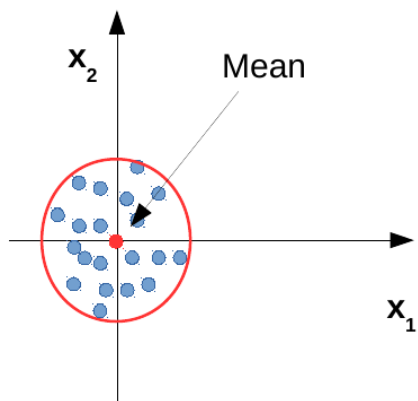
Multivariate Gaussians (V)

- Assume for a moment that $E[\ ] = 0 \rightarrow$ Covariance $E[x_1 x_2]$ is a "dot" product.

- Assume two points $[1\ 0]\begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1 \rightarrow$ Covariance measure similarity.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} 1 & 0.6 \\ \boxed{0.6} & 1 \end{bmatrix} \right)$$

$E[x_1 x_2] \neq 0$

$\rightarrow$ Knowing about x_1 DOES provide information about x_2 .

$\rightarrow$ if x_1 is positive, x_2 is with great probability.

$\rightarrow$ knowing something about x_1 allows us to know something about x_2.
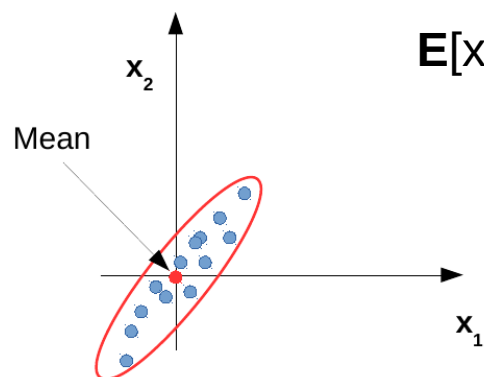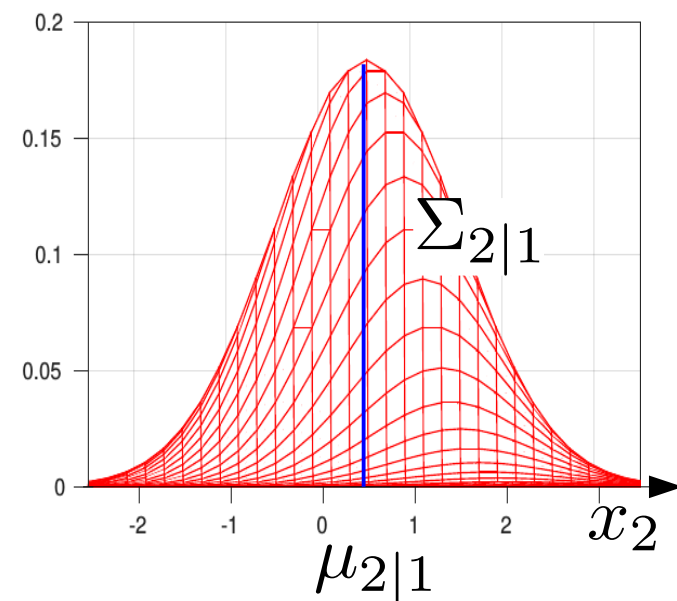
Joint Gaussian distributions

see, e.g., Rasmussen et al. (2005), Murphy (2012)

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N \left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right)$$
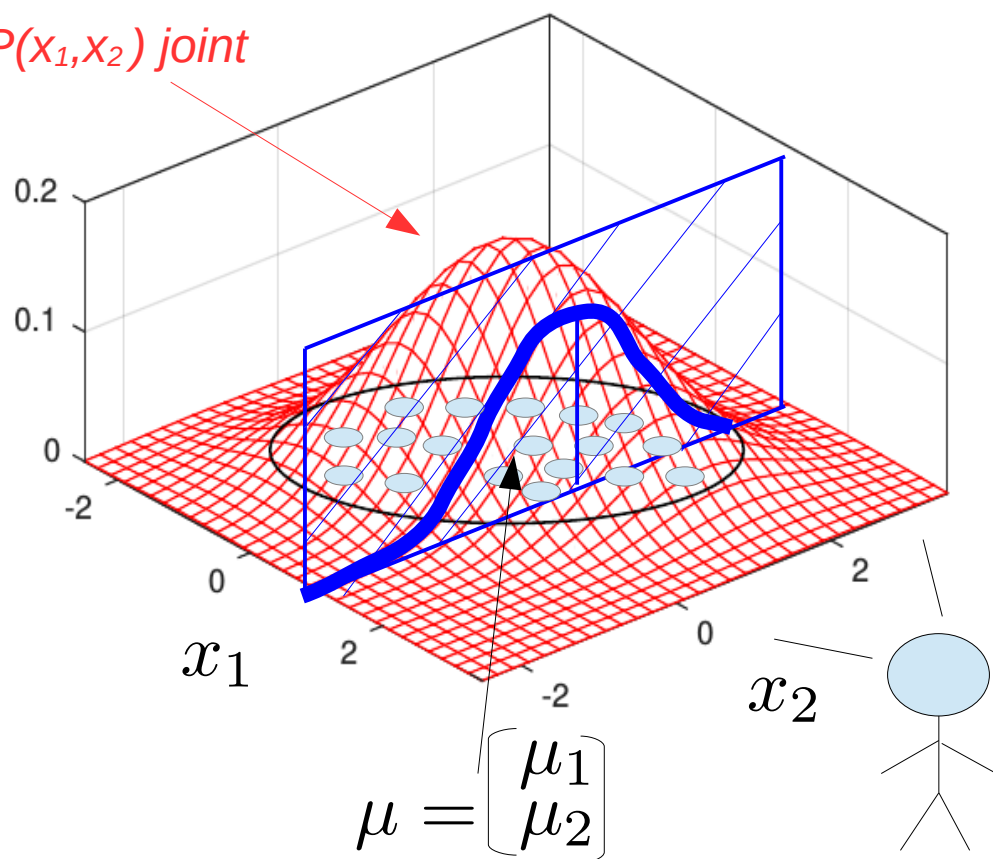
Mean   Covariance

Conditional distribution

$$P(x_2 | X_1 = x_1)$$

$P(x_1, x_2)$ joint

x_1

x_2

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$$

$\Sigma_{2|1}$

$\mu_{2|1}$

x_2

From Joint to Conditional distributions

see, e.g., Murphy (2012), chapter 4.

Theorem 4.3.1 (Marginals and conditionals of an MVN). *Suppose* $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$ *is jointly Gaussian with parameters*

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix} \tag{4.67}$$

Then the marginals are given by

$$\begin{aligned} p(\mathbf{x}_1) &= \mathcal{N}(\mathbf{x}_1 | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \\ p(\mathbf{x}_2) &= \mathcal{N}(\mathbf{x}_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \end{aligned} \tag{4.68}$$

and the posterior conditional is given by

Two "blocks" of vectors

$$\begin{aligned} p(\mathbf{x}_1 | \mathbf{x}_2) &= \mathcal{N}(\mathbf{x}_1 | \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\ \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12} \boldsymbol{\Sigma}_{22}^{-1} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1} \boldsymbol{\Lambda}_{12} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\Sigma}_{1|2} \left(\boldsymbol{\Lambda}_{11} \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{12} (\mathbf{x}_2 - \boldsymbol{\mu}_2) \right) \\ \boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12} \boldsymbol{\Sigma}_{22}^{-1} \boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1} \end{aligned} \tag{4.69}$$

This Theorem allows you to go from joint to conditional distributions.

Producing Data from Gaussians

$$x_i \sim \mathcal{N}(0, 1)$$

$$x_i \sim \mathcal{N}(\mu, \sigma^2) \sim \mu + \sigma \mathcal{N}(0, 1)$$

As we have the capability of drawing *1-dim* random numbers from a Gaussian, we can also do this in a multivariate case.
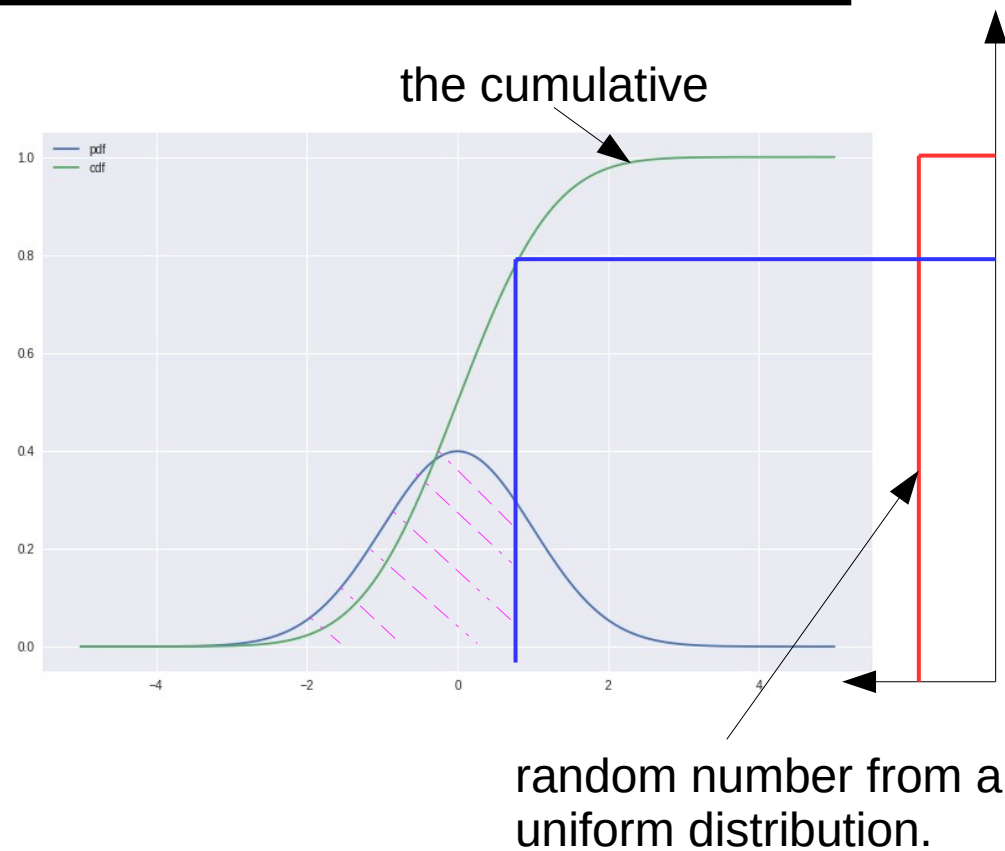
→ We need a way to take "square roots" from matrices.
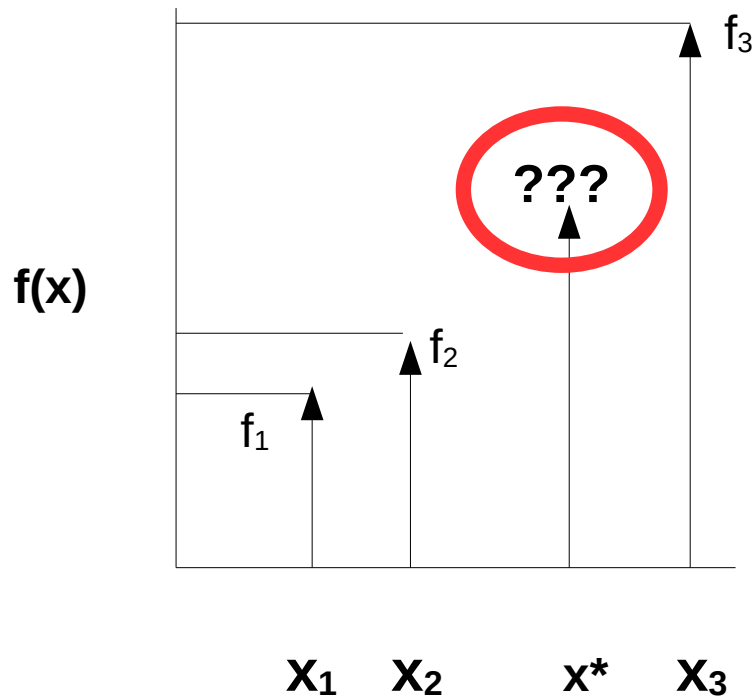
→ **Cholesky** decomposition: $\Sigma = LL^T$

the cumulative

random number from a uniform distribution.

$$\longrightarrow \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim N\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \right) \longrightarrow x \sim \mu + L\mathcal{N}(0, I)$$

$\mathbf{x}$ $\quad \mu \quad$ Σ

Recall Eq. (4.68) from the previous slide,
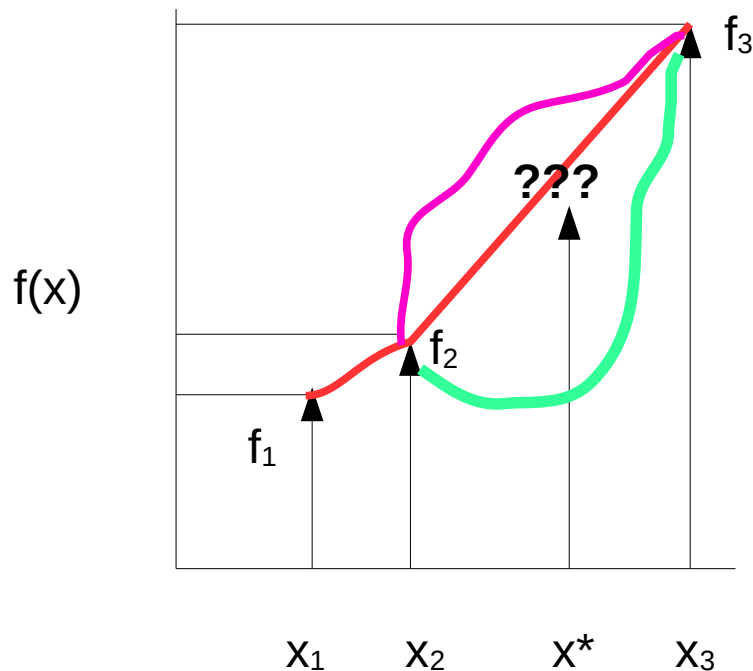
Observations → Interpolation

We have 3 observations at x_i for $f(x_i)$

→ **Given the data pairs**
$D = \{ (x_1,f_1), (x_2,f_2), (x_3,f_3)\}$

→ **want to find/learn the function that describes the data, i.e., for a "new" x*, we want to know what f(x*) would be!**

f(x)

f₃

???

f₂

f₁

X₁   X₂   x*   X₃

Observations → Interpolation (II)

We assume that **f's (the height) are Gaussian distributed**,
with **zero – mean** and some **covariance matrix K**.

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \sim N \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{11}, K_{12}, K_{13} \\ K_{21}, K_{22}, K_{23} \\ K_{31}, K_{32}, K_{33} \end{bmatrix} \right)$$
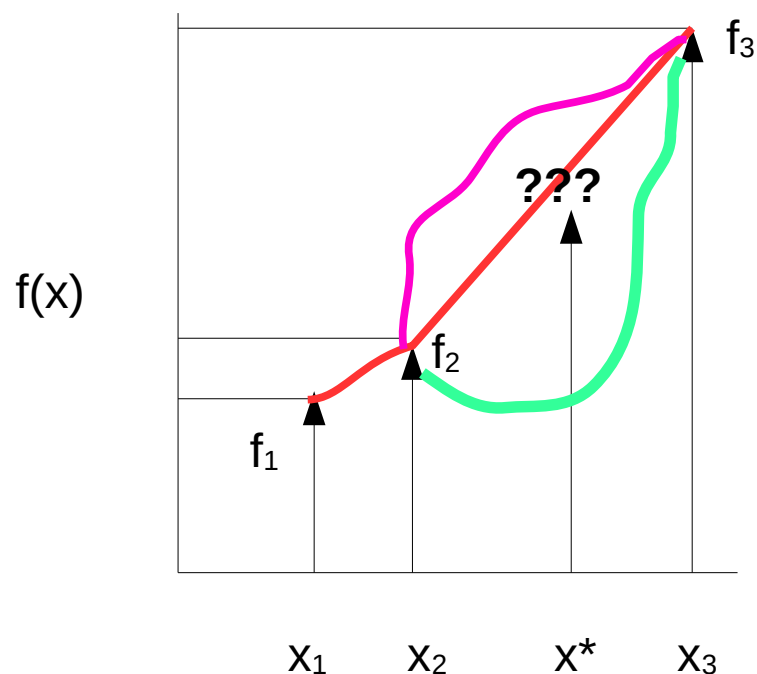
Note: f_1 and f_2 should probably be more correlated,
as they are nearby (compared to f_1 and f_3).

→ The prior mean function μ reflects the expected
function value at input x: $\mu(x) = \mathbb{E}(f(x))$

→ It is often set to 0.

Observations → Interpolation (II)

We assume that **f's (the height) are Gaussian distributed**, with **zero − mean** and some **covariance matrix K**.

f(x)

x_1 x_2 x^* x_3

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \sim N\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{11}, K_{12}, K_{13} \\ K_{21}, K_{22}, K_{23} \\ K_{31}, K_{32}, K_{33} \end{bmatrix}\right)$$

Note: f_1 and f_2 should probably be more correlated, as they are nearby (compared to f_1 and f_3), e.g.,

$$\sim N\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1, 0.7, 0.2 \\ 0.7, 1, 0.6 \\ 0.2, 0.6, 1 \end{bmatrix}\right)$$

Covariance matrix **constructed** by some "**measure of similarity**", i.e., a **kernel function** (parametric ansatz), such as "squared exponential". Parameters can be obtained e.g. via MLE (later).

$$\kappa(x, x') = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x - x')^2\right)$$

σ_f^2 – controls vertical variation.

ℓ – controls horizontal length scale.

Observations → Interpolation (III)

Given data D = { (x_1, f_1), (x_2, f_2), (x_3, f_3)} → **f(x*) = f*** ?
→ Assume f ~ $N(0, K(\cdot, \cdot))$
→ Assume f(x*) ~ $N(0, K(x^*, x^*))$

**3d-Covariance K from the training data**

f(x)

f₃
f₂
f₁

???
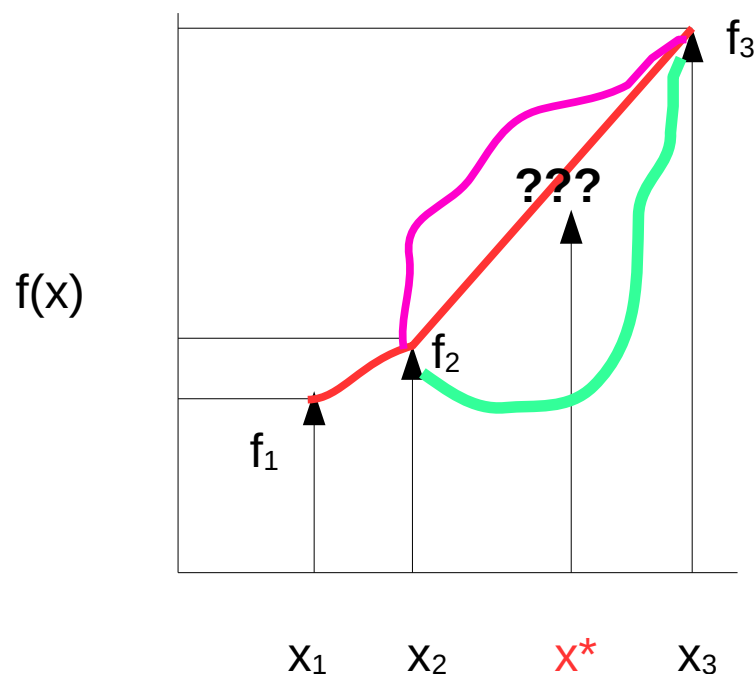
$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_* \end{pmatrix} \sim N \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} K_{11}, K_{12}, K_{13}, K_{1*} \\ K_{21}, K_{22}, K_{23}, K_{2*} \\ K_{31}, K_{32}, K_{33}, K_{3*} \\ K_{*1}, K_{*2}, K_{*3}, K_{**} \end{bmatrix} \right)$$

x₁   x₂   x*   x₃

→ **Joint distribution over f and f*** .
→ **We need the conditional of f* given f**.
→ In this example, we "cut" in 3 dimensions.
→ What is left is a 1-dimensional Gaussian,
   i.e., the Gaussian for f*

$$K(x_1, x_*) = K_{1*}$$

Interpolation → Noiseless GPR

(see, e.g., Rasmussen & Williams (2006), with references therein)
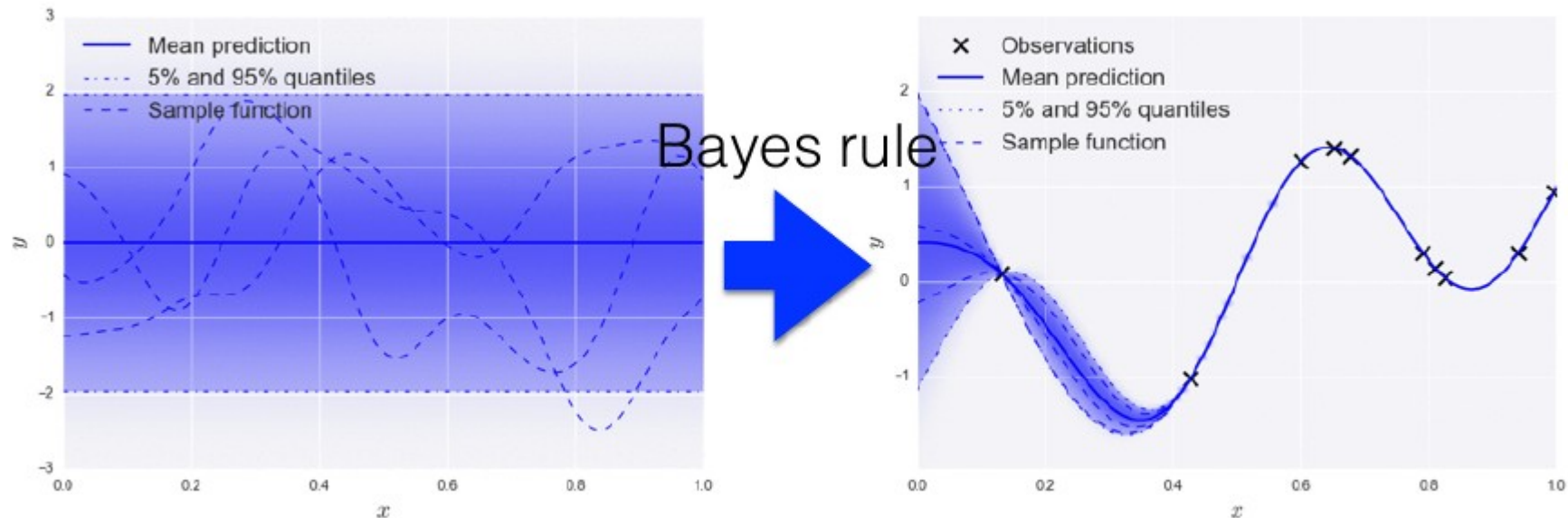
Prior GP → Bayes rule → Posterior GP

Training set: $D = \{(\mathbf{x}_i, y_i) | i = 1, \ldots, n\}$

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right)$$

$$p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{f}) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$$

$$\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{X}_*) + \mathbf{K}_*^T \mathbf{K}^{-1}(\mathbf{f} - \boldsymbol{\mu}(\mathbf{X}))$$

$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*$$

Test point = interpolation at X*

→ **predictive mean** $\mu_* = \mathbb{E}(f_*)$

→ **Confidence Intervals!**
Where we have data, we have high confidence in our predictions.
Where we do not have data, we cannot be to confident about our predictions.

Algorithmic Complexity!

- The central computational operation in using Gaussian processes will involve the inversion of a matrix of size N × N, for which standard methods require $O(N^3)$ computations.

- The matrix inversion must be performed once for the given training set.

  → For large training data sets, however, the direct application of Gaussian process methods can become infeasible.

  → Sparse Methods (cf. Rasmussen et. al (2006) and references therein).

A note on the predictive mean

Note that the predictive mean can (in general) also be written as

$$\mu = m(x) + \sum_{i=1}^{N} a_i k(x_i, x)$$

where $a = (a_1, \ldots, a_N) = \left(\mathbf{K} + s_n^2 \mathbf{I}_N\right)^{-1} (\mathbf{t} - \mathbf{m})$
and *t* being the *N* observations.

→ We can think of the GP posterior mean as an approximation of f(·) using *N* symmetric basis functions centered at each observed input.

→ by choosing a covariance function that vanishes when x and x' are separated by a lot, for example the squared exponential covariance function, we see that an observed input-output will only affect the approximation locally.

$$k_{\mathrm{SE}}(\mathbf{x}, \mathbf{x}'; \boldsymbol{\theta}) = s^2 \exp\left\{-\frac{1}{2} \sum_{i=1}^{D} \frac{(x_i - x_i')^2}{\ell_i^2}\right\}$$

GP – a distribution over functions

day2/code/1d_gp_example.ipynb

$$f(x) \sim GP(\mu(x), k(x, x'))$$

$$\mu(x) = \mathbb{E}[f(x)]$$

$$k(x, x') = \mathbb{E}[(f(x) - \mu(x))((f(x') - \mu(x'))^T]$$

$$k(x, x') = exp(-\frac{1}{2}(x - x')^2)$$

Procedure
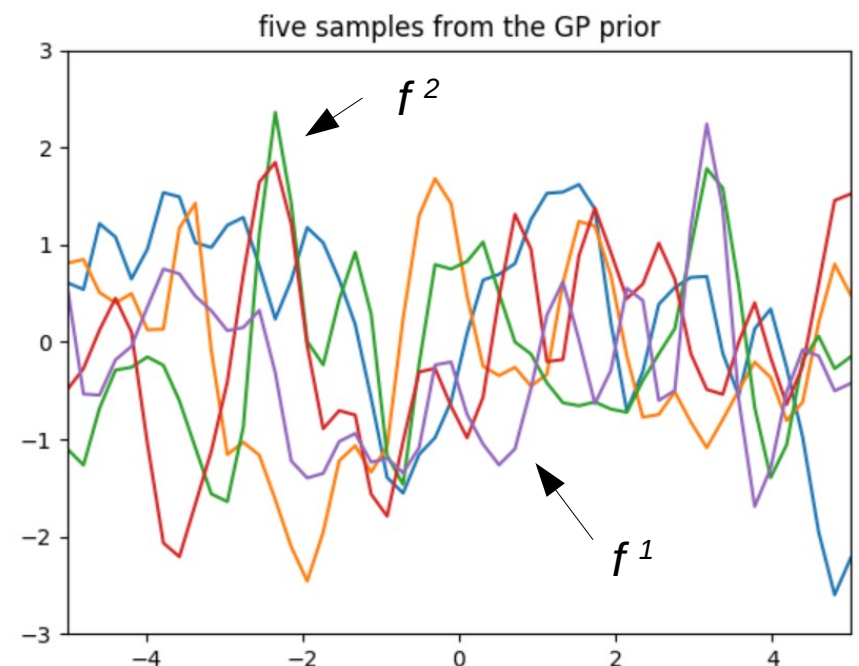Create vector $X_{1:N}$

$$\mu = 0, K_{N \times N}$$

$K = LL^T$

$f^i \sim N(0, K) \sim L\, N(0, \mathbf{I})$

five samples from the GP prior

f^2

f^1

GPR Example code & numerical stability

Look at demo/1d_gp_example.ipynb and Murphy, Chapter 15

```python
import numpy as np
import matplotlib.pyplot as pl

""" A code to illustrate the workings of GP regression in 1d.
    We assume a zero mean GP Prior """


# This is the true unknown, one-dimensional function we are trying to approximate
f = lambda x: np.sin(0.9*x).flatten()


# This is the kernel function
def kernel_function(a, b):
    """ GP squared exponential kernel function """
    kernelParameter = 0.1
    sqdist = np.sum(a**2,1).reshape(-1,1) + np.sum(b**2,1) - 2*np.dot(a, b.T)
    return np.exp(-.5 * (1/kernelParameter) * sqdist)


# Here we run the test
N = 10          # number of training points.
n = 50          # number of test points.
s = 0.00005     # noise variance.

# Sample some input points and noisy versions of the function evaluated at
# these points.
X = np.random.uniform(-5, 5, size=(N,1))
y = f(X) + s*np.random.randn(N)   #add some noise

K = kernel_function(X, X)
L = np.linalg.cholesky(K + s*np.eye(N))

# points we're going to make predictions at.
TestPoint = np.linspace(-5, 5, n).reshape(-1,1)

# compute the mean at our test points.
Lk = np.linalg.solve(L, kernel_function(X, TestPoint))
mu = np.dot(Lk.T, np.linalg.solve(L, y))

# compute the variance at our test points.
K_ = kernel_function(TestPoint, TestPoint)
s2 = np.diag(K_) - np.sum(Lk**2, axis=0)
s = np.sqrt(s2)
```

$$\mathbb{E}[f(x_*)] = \mu = k_*^T K_y^{-1} y$$
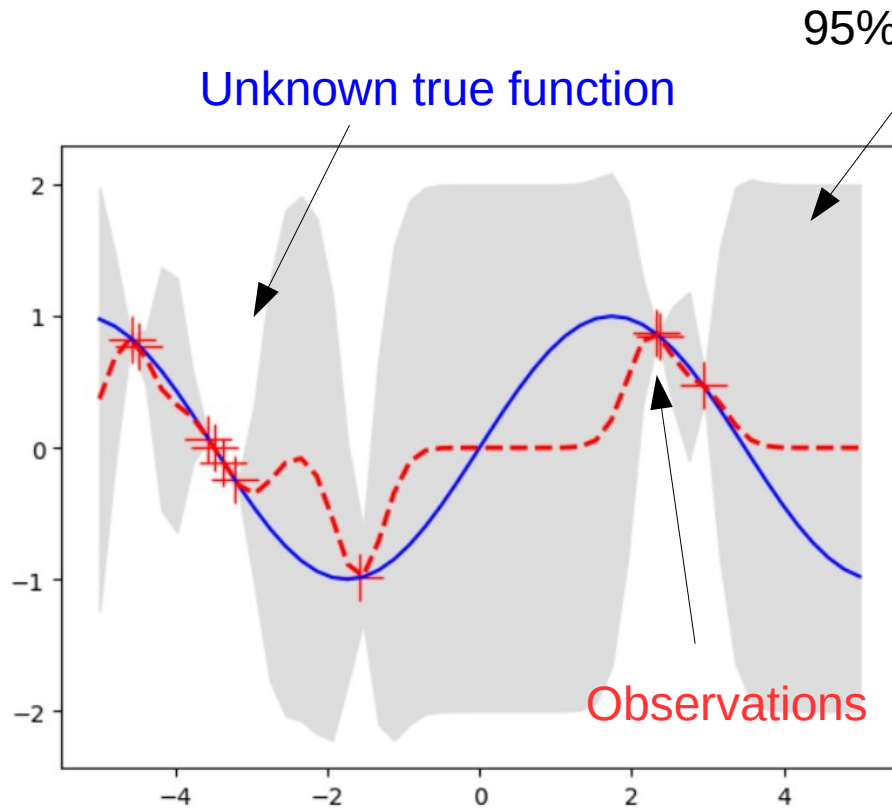
$$K_y = LL^T$$

$$\alpha = K_y^{-1} y = L^{-T} L^{-1} y$$

Algorithm 15.1: GP regression

1 $\mathbf{L} = \text{cholesky}(\mathbf{K} + \sigma_y^2 \mathbf{I});$

2 $\alpha = \mathbf{L}^T \setminus (\mathbf{L} \setminus \mathbf{y});$

3 $\mathbb{E}[f_*] = \mathbf{k}_*^T \alpha$;

4 $\mathbf{v} = \mathbf{L} \setminus \mathbf{k}_*;$

5 $\text{var}[f_*] = \kappa(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{v}^T \mathbf{v};$

6 $\log p(\mathbf{y}|\mathbf{X}) = -\frac{1}{2}\mathbf{y}^T \alpha - \sum_i \log L_{ii} - \frac{N}{2} \log(2\pi)$

Alg. 15: Numerical more stable.

Prediction & Confidence Intervals

95% Confidence Intervals

Unknown true function

Observations

10 Training Points
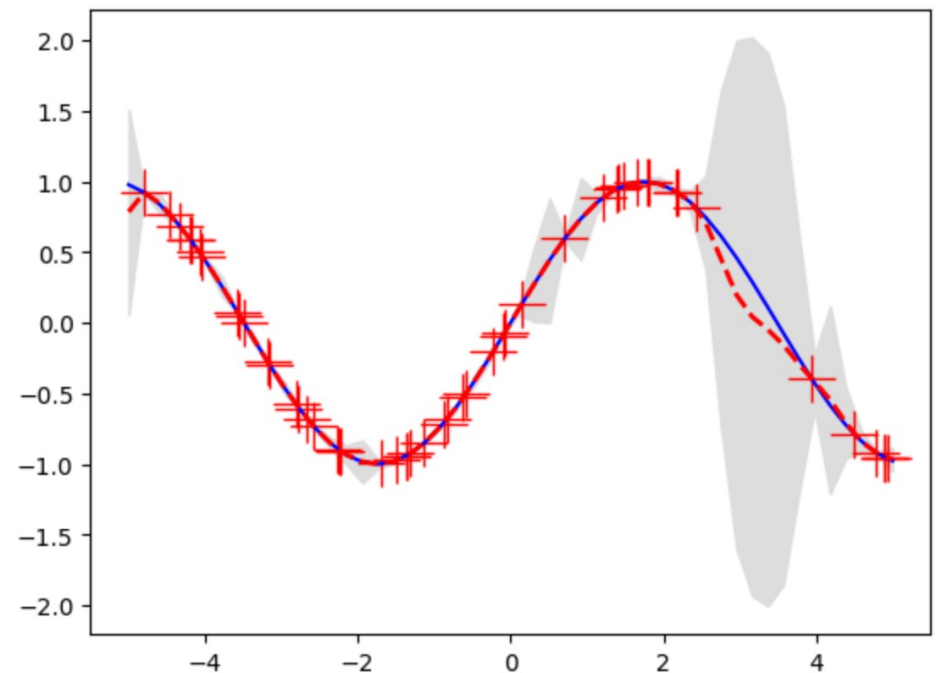
50 Training Points

We see that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.

Non-hypercubic domain: GPR versus ASG

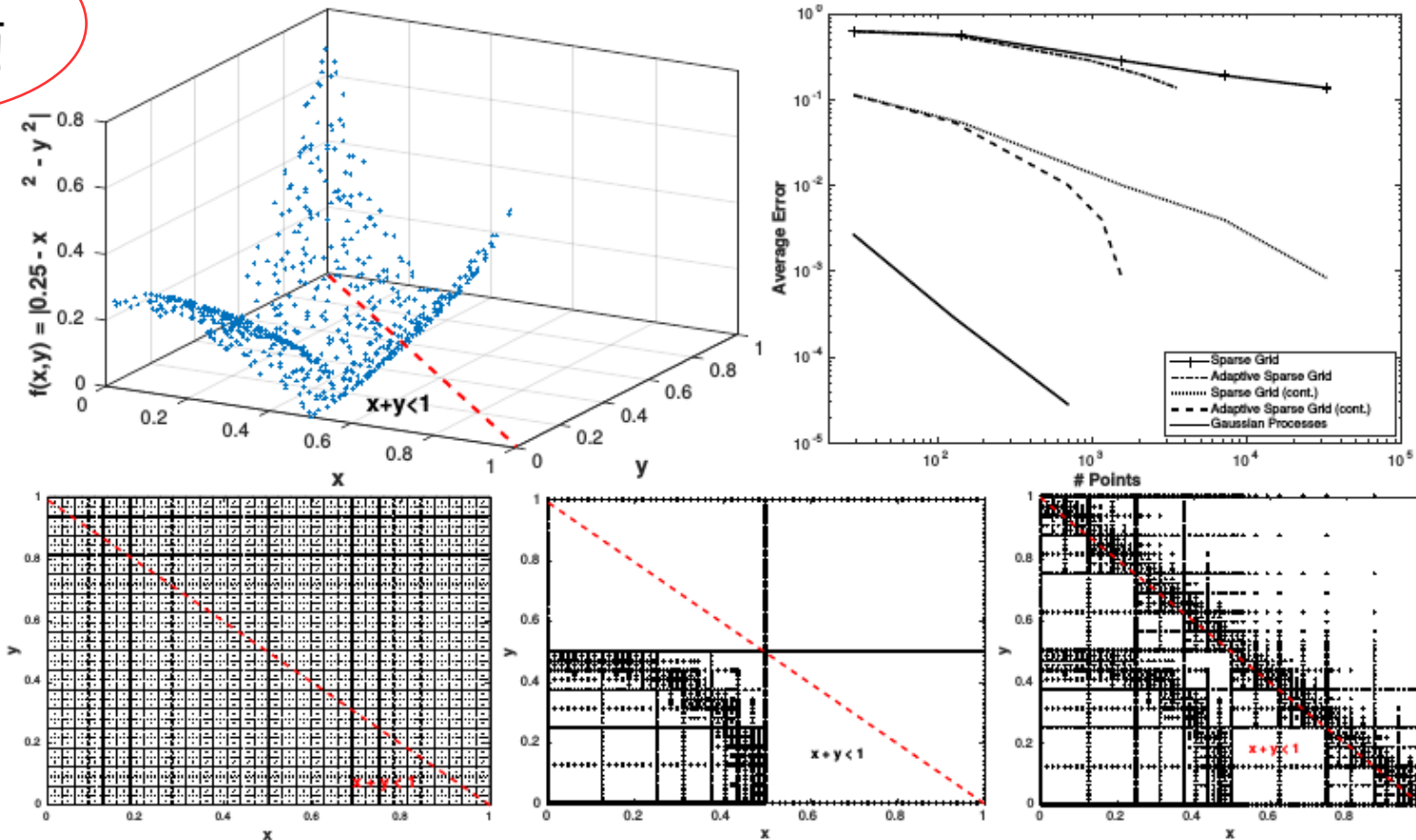$$\text{Vol}_\Delta = \frac{1}{D!}$$

Figure: The upper left panel shows the analytical test function evaluated at random test points on the simplex. The upper right panel displays a comparison of the interpolation error for GPs, sparse grids, and adaptive sparse grids of varying resolution and constructed under the assumption that a continuation value exists, (denoted by cont"), or that there is no continuation value. The lower left panel displays a sparse grid consisting of 32,769 points. The lower middle panel shows an adaptive sparse grid (cont) that consists of 1,563 points, whereas the lower right panel shows an adaptive sparse grid, constructed with 3,524 points and under the assumption that the function outside Δ is not known.

GPR in scikit-learn.org

https://scikit-learn.org/stable/modules/gaussian_process.html
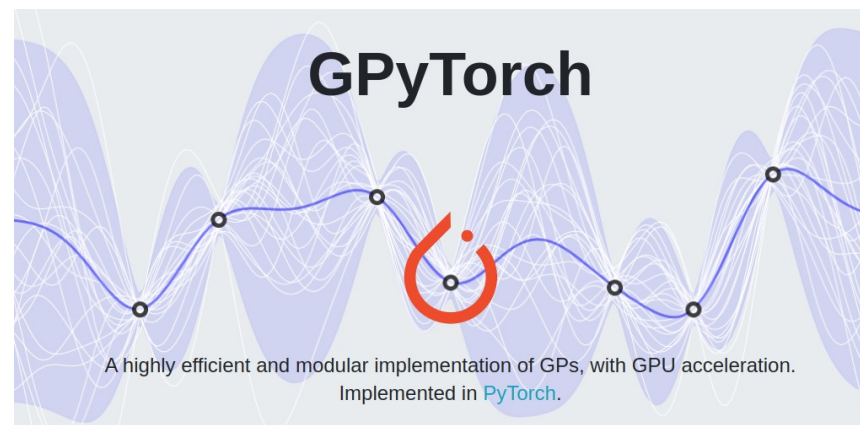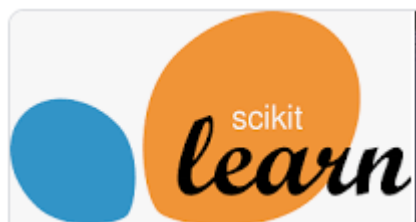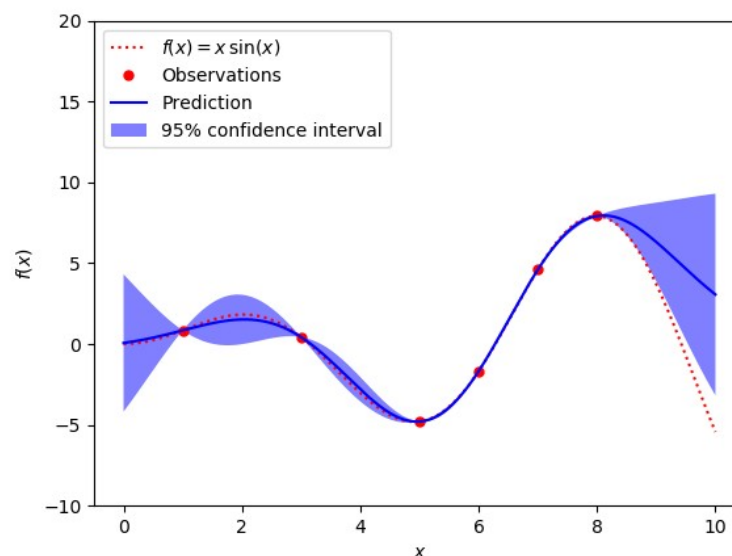
Look at demo/1d_GPR.ipynb

Illustration of noiseless GPR prediction (II)

- We use a squared exponential kernel, aka Gaussian kernel or RBF kernel.

- In *1d*, this is given by $$\kappa(x, x') = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x - x')^2\right)$$

- Here *l* controls the horizontal length scale over which the function varies, and σ_f controls the vertical variation.

- We usually show predictions from the posterior, *p(f* $_*$ *|X* $_*$ *, X, f)*.

- We see (NEXT SLIDE) that the model perfectly interpolates the training data, and that the predictive uncertainty increases as we move further away from the observed data.
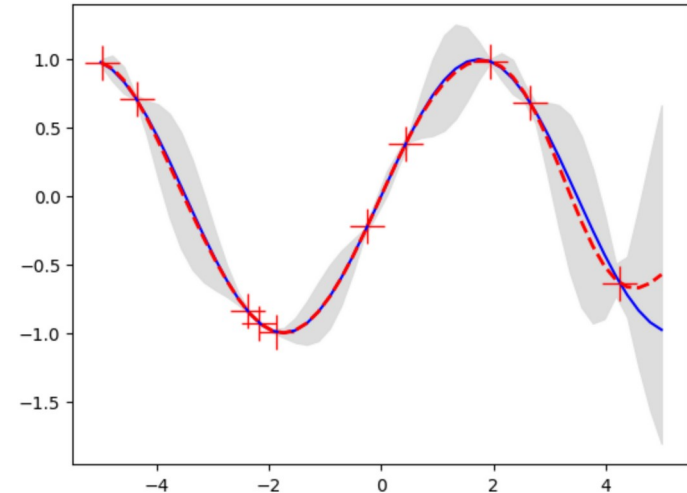
The parameters in the Kernel

cf. demo/1d_gp_example.ipynb

$$\kappa(x, x') = \sigma_f^2 \exp(-\frac{1}{2\ell^2}(x - x')^2)$$
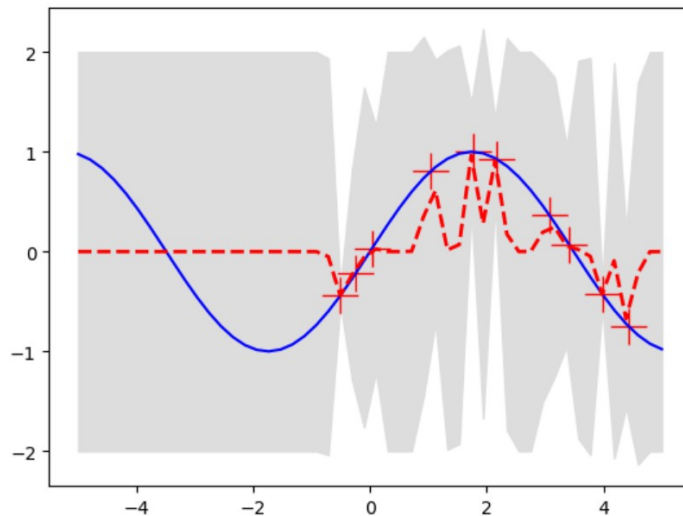
Let $\sigma_f^2 = 1$

→ **Tuning the parameters by hand
is not a good idea in general
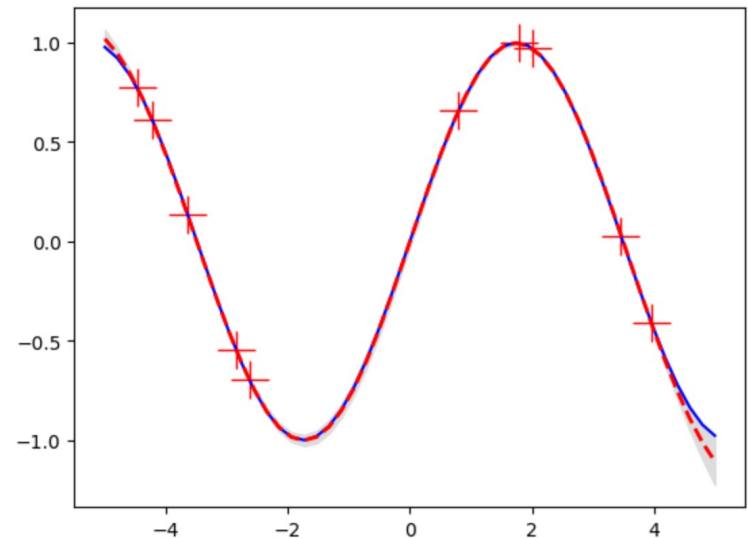(in particular in high-dimensional settings)**.

l^2 = 1.0

l^2 = 0.01

l^2 = 10.0

GPR with noisy data

- In empirical setups, measurement noise may arise from our inability to control all the influential factors or from irreducible (aleatory) uncertainties.

- In computer simulations, measurement uncertainty may stem from quasi-random stochasticity, or chaotic behavior.

- Now let us consider the case where what we observe is a noisy version of the underlying function

$$y = f(x) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma_y^2)$$

GPR with noisy data (II)

- In this case (presence of noise), the model is not required to interpolate the data, but it must come "close" to the observed data.

- The covariance of the observed noisy responses is

$$\text{cov}\left[y_p, y_q\right] = \kappa(\mathbf{x}_p, \mathbf{x}_q) + \sigma_y^2 \delta_{pq}$$ where $\delta_{pq} = \mathbb{I}(p = q)$

- The second matrix is diagonal because we assumed the noise terms were independently added to each observation.

The GPR with noisy data (III)

- The joint density of the observed data and the **latent, noise-free function** on the test points is given by

Noise in the diagonal, rest is as before.

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K}_y & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right)$$

Latent function.

- where we are assuming the mean is zero, for notational simplicity.

- Hence the posterior predictive density is

$$
\begin{aligned}
p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{y}) &= \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \\
\boldsymbol{\mu}_* &= \mathbf{K}_*^T \mathbf{K}_y^{-1} \mathbf{y} \\
\boldsymbol{\Sigma}_* &= \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}_y^{-1} \mathbf{K}_*
\end{aligned}
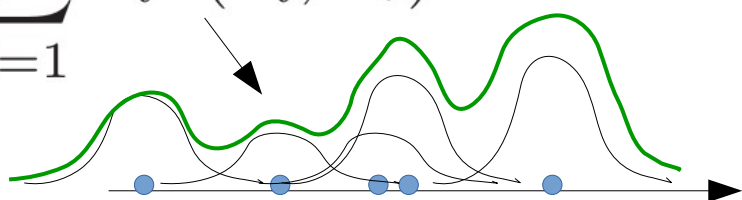$$

Prediction at a single test point

In the case of a single test input, this simplifies as follows

$$p(f_* | \mathbf{x}_*, \mathbf{X}, \mathbf{y}) \;=\; \mathcal{N}(f_* | \mathbf{k}_*^T \mathbf{K}_y^{-1} \mathbf{y}, \, k_{**} - \mathbf{k}_*^T \mathbf{K}_y^{-1} \mathbf{k}_*)$$

where $\quad \mathbf{k}_* \;=\; \big[\kappa(\mathbf{x}_*, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}_*, \mathbf{x}_N)\big]$
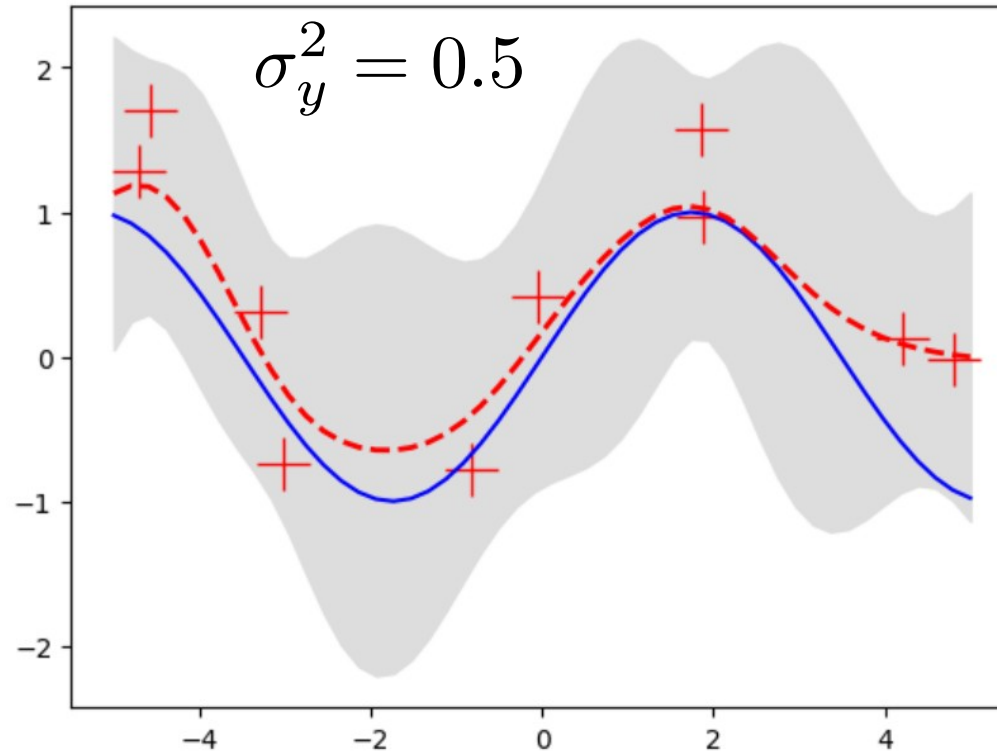
and where $\quad k_{**} \;=\; \kappa(\mathbf{x}_*, \mathbf{x}_*) \quad$ (=1)

Again, we can write the posterior mean as expansion of basis functions

1 x N

N x 1

$$\overline{f}_* = \mathbf{k}_*^T \mathbf{K}_y^{-1} \mathbf{y} = \sum_{i=1}^{N} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x}_*) \quad \text{where } \boldsymbol{\alpha} = \mathbf{K}_y^{-1} \mathbf{y}$$

N x N

from training data

Some Plots

cf. demo/1d_gp_example.ipynb

$$\sigma_y^2 = 0.5$$

- Even in the regions where you have data, there is still uncertainty.

- In the noise-free version of GPR, the uncertainty is 0 at observation points.

- But we still have the same properties as before: where we have data, we are more certain compared to the case where we have no data.

Noise improves numerical stability

• It is common to use small noise even if there is not any in the data.

• Cholesky fails when covariance is close to being semi-positive definite.

• Adding a small noise improves numerical stability.

• It is known as the "jitter" or as the "nugget" in this case.

"Learning" the kernel parameters

- To estimate the kernel parameters, we could use exhaustive search over a discrete grid of values, with validation loss as an objective, but this can be quite slow.

- Here we consider an empirical Bayes approach, which will allow us to use continuous optimization methods, which are much faster.

- In particular, we will maximize the marginal likelihood.

"Learning" the kernel parameters (II)

Marginal likelihood $\quad p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{f}, \mathbf{X}) p(\mathbf{f}|\mathbf{X}) d\mathbf{f}$

Since $\quad p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\mathbf{0}, \mathbf{K})$

and $\quad p(\mathbf{y}|\mathbf{f}) = \prod_i \mathcal{N}(y_i|f_i, \sigma_y^2)$

the (log-) marginal likelihood is given by

$$\log p(\mathbf{y}|\mathbf{X}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{K}_y| - \frac{N}{2}\log(2\pi)$$

1st term: data fit term
2nd term: a model complexity term
3rd term: a constant.

Maximizing the the marginal likelihood

- Let the kernel parameters (also called hyper-parameters) be denoted by $\boldsymbol{\theta}$

- One can show that the following holds.

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} \log p(\mathbf{y}|\mathbf{X}) &= \frac{1}{2}\mathbf{y}^T \mathbf{K}_y^{-1} \frac{\partial \mathbf{K}_y}{\partial \theta_j} \mathbf{K}_y^{-1} \mathbf{y} - \frac{1}{2}\operatorname{tr}(\mathbf{K}_y^{-1} \frac{\partial \mathbf{K}_y}{\partial \theta_j}) \\
&= \frac{1}{2}\operatorname{tr}\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^T - \mathbf{K}_y^{-1}) \frac{\partial \mathbf{K}_y}{\partial \theta_j} \right)
\end{aligned}
$$

  where $\boldsymbol{\alpha} = \mathbf{K}_y^{-1}\mathbf{y}$

Computational complexity:
- It takes $O(N^3)$ time to compute $\mathbf{K}_y^{-1}$

- $O(N^2)$ time per hyper-parameter to compute the gradient.

Careful: Different optima correspond to different interpretations/believes

Fig. 5.5 of (Rasmussen and Williams 2006).

- We use the SE kernel

$$\kappa_y(x_p, x_q) = \sigma_f^2 \exp(-\frac{1}{2\ell^2}(x_p - x_q)^2) + \sigma_y^2 \delta_{pq}$$

- with $\sigma_f^2 = 1$

- We plot $\log p(\mathbf{y}|\mathbf{X}, \ell, \sigma_y^2)$ (where X and y are the 7 data points shown) as we vary ℓ and σ_y^2.

- The two local optima are indicated by +.

- The bottom left optimum corresponds to a low-noise, short-length scale solution.

- The top right optimum corresponds to a high-noise, long-length scale solution.

- With only 7 data points, there is not enough evidence to confidently decide which is more reasonable.
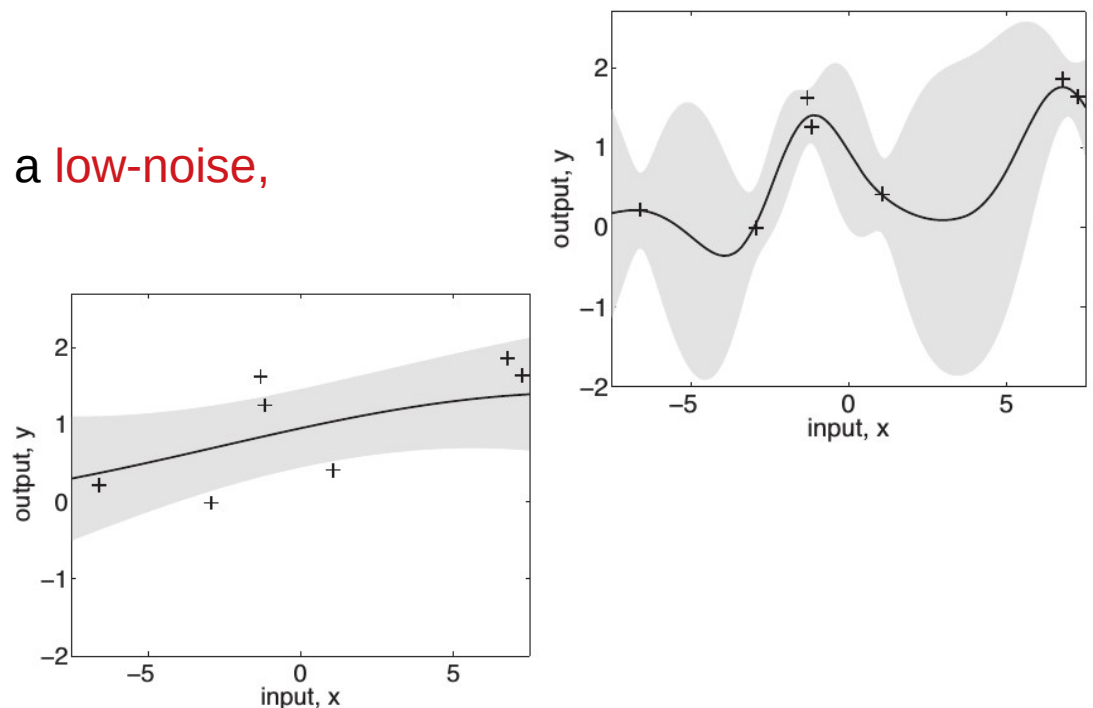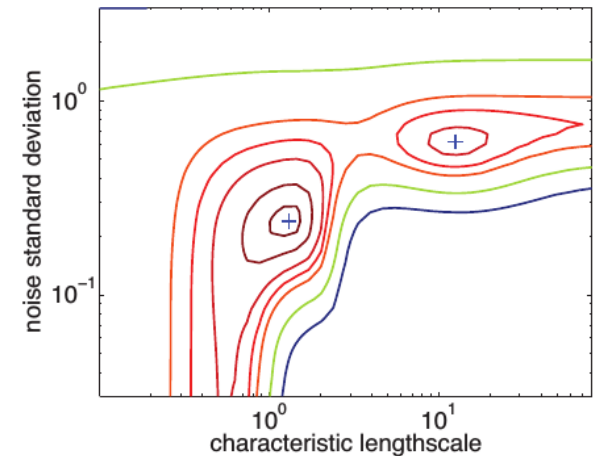
An example: noise and optimization

demo/GPR_scikit_noise.ipynb

```python
import numpy as np
from matplotlib import pyplot as plt

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C

np.random.seed(1)


def f(x):
    """The function to predict."""
    return x * np.cos(x)*np.sin(x)

# ----------------------------------------------------------------------
# Mesh the input space for evaluations of the real function, the prediction and
# its MSE
x = np.atleast_2d(np.linspace(0, 10, 1000)).T

# Instantiate a Gaussian Process model
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))

# now the noisy case
X = np.linspace(0.1, 9.9, 20)
X = np.atleast_2d(X).T

# Observations and noise
y = f(X).ravel()
dy = 0.5 + 1.0 * np.random.random(y.shape)
noise = np.random.normal(0, dy)
y += noise

# Instantiate a Gaussian Process model
gp = GaussianProcessRegressor(kernel=kernel, alpha=dy ** 2,
                              n_restarts_optimizer=10)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis (ask for MSE as well)
y_pred, sigma = gp.predict(x, return_std=True)

# Plot the function, the prediction and the 95% confidence interval based on
# the MSE
plt.figure()
plt.plot(x, f(x), 'r:', label=u'$f(x) = x\,\sin(x)$')
plt.errorbar(X.ravel(), y, dy, fmt='r.', markersize=10, label=u'Observations')
plt.plot(x, y_pred, 'b-', label=u'Prediction')
plt.fill(np.concatenate([x, x[::-1]]),
         np.concatenate([y_pred - 1.9600 * sigma,
                        (y_pred + 1.9600 * sigma)[::-1]]),
         alpha=.5, fc='b', ec='None', label='95% confidence interval')
plt.xlabel('$x$')
plt.ylabel('$f(x)$')
plt.ylim(-10, 20)
plt.legend(loc='upper left')

plt.show()
```

A multi-d example

demo/scikit_multi-d.ipynb

```python
import numpy as np
from matplotlib import pyplot as plt
import cPickle as pickle
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C


np.random.seed(1)


# Test function
def f(x):
    """The 2d function to predict."""
    return np.sin(x[0]) * np.cos(x[1])


# generate training data
n_sample = 100  #points
dim = 2         #dimensions

X = np.random.uniform(-1., 1., (n_sample, dim))
y = np.sin(X[:, 0:1]) * np.cos(X[:, 1:2]) + np.random.randn(n_sample, 1) * 0.005

# Instantiate a Gaussian Process model
kernel = C(1.0, (1e-3, 1e3)) * RBF(10, (1e-2, 1e2))
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9)

# Fit to data using Maximum Likelihood Estimation of the parameters
gp.fit(X, y)

# Make the prediction on the meshed x-axis / training points
y_pred, sigma = gp.predict(X, return_std=True)

#Compute MSE
mse = 0.0
n_sample_test=50
Xtest1 = np.random.uniform(-1., 1., (n_sample_test, dim))
y_pred1, sigma = gp.predict(Xtest1, return_std=True)
for g in range(len(Xtest1)):
    delta = abs(y_pred1[g] - f(Xtest1[g]))
    mse += delta

mse = mse/len(y_pred)
print(".......................")
print(" The MSE is ", mse[0])
print(".......................")
```

A multi-d example (II)

demo/scikit_multi-d.ipynb

```python
#-------------------------------------------------------------------

# Important -- save the model to a file
with open('2d_model.pcl', 'wb') as fd:
    pickle.dump(gp, fd, protocol=pickle.HIGHEST_PROTOCOL)
    print("data written to disk")


# Load the model and do predictions
with open('2d_model.pcl', 'rb') as fd:
    gm = pickle.load(fd)
    print("data loaded from disk")

# generate training data
n_test = 50
dim = 2
Xtest = np.random.uniform(-1., 1., (n_test, dim))
y_pred_test, sigma_test = gm.predict(Xtest, return_std=True)

MSE2 = 0
for a in range(len(Xtest)):
    delta = abs(y_pred_test[a] - f(Xtest[a]))
    MSE2 +=delta


MSE2 = MSE2/len(Xtest)
print(".........................")
print(" The MSE 2 is ", MSE2[0])
print(".........................")
#-------------------------------------------------------------------
```

More on Kernels

- Our prior beliefs about the response are encoded in our choice of the mean and covariance functions.

- The choice of an appropriate kernel is based on assumptions such as smoothness and likely patterns to be expected in the data.

- A sensible assumption is usually that the correlation between two points decays with distance between the points according to some power function.

- The choice of kernel determines almost all the generalization properties of a GP model.

- **You are the expert on your modeling problem - so you're the person best qualified to choose the kernel!**

Stationary versus non-stationary Kernels

Two categories of kernels can be distinguished:

- **Stationary kernels**:
  They depend only on the distance of two data points and not on their absolute values
  $k(x_i, x_j) = k(d(x_i, x_j))$ and are thus invariant to translations in the input space

  Stationary kernels can further be subdivided into isotropic and anisotropic kernels, where isotropic kernels are also invariant to rotations in the input space.

- **Non-stationary kernels**:
  They depend also on the specific values of the data points.

Squared Exponential Kernel

The SE kernel has become the **de-facto default kernel** for GPs.

$$k_{\mathrm{SE}}(x, x') = \sigma^2 \exp\left(-\frac{(x-x')^2}{2\ell^2}\right)$$

This is probably because it has some nice properties:

- It is universal, and you can integrate it against most functions that you need to.

- Every function in its prior has infinitely many derivatives.

- It also has only two parameters:
  - The **lengthscale ℓ** determines the length of the 'wiggles' in your function. In general, **you won't be able to extrapolate more than ℓ units away from your data**.
  - The output **variance σ²** determines the average distance of your function away from its mean. Every kernel has this parameter out in front; it's just a scale factor.

Pitfalls for the SE kernel

- Most people who set up a GP regression (or classification) model end up using the SE kernel.

- They are a quick-and-dirty solution that will probably work pretty well for interpolating smooth functions when N is a multiple of D, and when there are no 'kinks' in your function.

- If your function happens to have a discontinuity or is discontinuous in its first few derivatives (for example, the abs() function), then either your **length-scale** will end up being **extremely short**, and your posterior mean will become zero almost everywhere, or your posterior mean will have 'ringing' effects.

- Even if there are no hard discontinuities, the length-scale will usually end up being determined by the smallest 'wiggle' in your function - so you might end up failing to extrapolate in smooth regions if there is even **a small non-smooth** region in your data.

- If your data is more than two-dimensional, it may be hard to detect this problem. One indication is if the length-scale chosen by maximum marginal likelihood never stops becoming smaller as you add more data. This is a classic sign of **model misspecification**.

Periodic Kernel
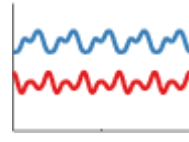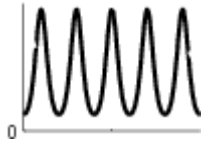
$$k_{\mathrm{Per}}(x, x') = \sigma^2 \exp\left(-\frac{2\sin^2(\pi|x - x'|/p)}{\ell^2}\right)$$

- The periodic kernel allows one to model functions which repeat themselves exactly.

- Its parameters are easily interpretable:

  - The period **p** simply determines the **distance between repetitions** of the function.

  - The **length-scale ℓ** determines the length-scale function in the same way as in the SE kernel.
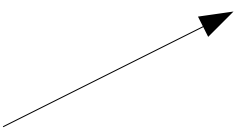
Matérn kernel

- The **Matérn kernel** is a stationary kernel and a generalization of the RBF kernel.

- It has an additional parameter *v* which controls the smoothness of the resulting function. It is parameterized by a length-scale parameter ℓ >0, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs (anisotropic variant of the kernel).

- The kernel is given by:

$$k(x_i, x_j) = \sigma^2 \frac{1}{\Gamma(\nu)2^{\nu-1}} \left(\gamma\sqrt{2\nu}d(x_i/l, x_j/l) \right)^{\nu} K_\nu \left(\gamma\sqrt{2\nu}d(x_i/l, x_j/l) \right)$$

Matérn kernel (II)

- As $v \to 0$, the Matérn kernel converges to the RBF kernel.

- When $v = 1/2$, the Matérn kernel becomes identical to the absolute exponential kernel.

$$k_{\mathrm{mat}}(x, x') = \sigma^2 \left(1 + \sqrt{3} \sum_{i=1}^{l} \frac{(x_i - x_i')^2}{\ell_i^2}\right) \exp\left(-\sqrt{3} \sum_{i=1}^{l} \frac{(x_i - x_i')^2}{\ell_i^2}\right)$$

- This are popular choices for learning functions that are not infinitely differentiable (as assumed by the RBF kernel) but at least once ($v = 3/2$) (if $v = 5/2$, the kernel is twice differentiable).

Combining/Adding Kernels

- Roughly speaking, adding two kernels can be thought of as an **OR** operation.

  → If you add together two kernels, then the resulting kernel will have high value if either of the two base kernels have a high value.

- **Linear plus Periodic Kernel**

  - A linear kernel plus a periodic results in functions which are periodic with increasing mean as we move away from the origin.

- **Adding across dimensions**

  - Adding kernels which each depend only on a single input dimension results in a prior over functions which are a sum of one-dimensional functions, one for each dimension. That is, the function f(x,y) is simply a sum of two functions $f_x(x) + f_y(y)$

  - These kernels have the form:

    $$k_{\mathrm{additive}}(x, y, x', y') = k_x(x, x') + k_y(y, y')$$
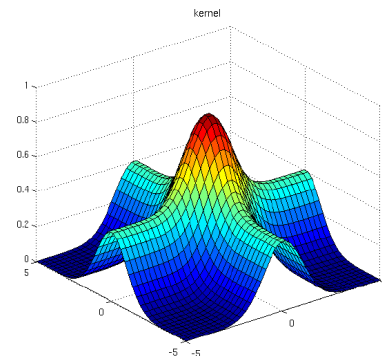
Automatically choosing Kernels

+ Sometimes, it is not obvious which kernel is appropriate for your problem.

+ In fact, you might decide that choosing the kernel is one of the main difficulties in doing inference

+ Just as you don't know what the true parameters are, you also don't know what the true kernel is.

+ Probably, you should try out a few different kernels at least, and compare their marginal likelihood on your training data.

+ Automatic ways:
  + https://arxiv.org/abs/1302.4922
    (Structure Discovery in Nonparametric Regression through Compositional Kernel Search)
  + https://github.com/jamesrobertlloyd/gp-structure-search

Two example motebooks

- lectures/day3/code/gpytorch_tutorial.ipynb

- lectures/day3/code/gpytorch_tutorial.ipynb

Pricing options with GPs (BS surrogate model)

- see day3/code/GP-BS-Pricing_01.ipynb

Greeks

- see day3/code/GP-BS-Pricing_02.ipynb

- The GP provides analytic derivatives with respect to the input variables

$$\partial_{X_*}\mathbb{E}[\mathbf{f}_*|X, Y, X_*] = \partial_{X_*}\boldsymbol{\mu}_{X_*} + \partial_{X_*}K_{X_*,X}\alpha$$

$$\partial_{X_*}K_{X_*,X} = \frac{1}{\ell^2}(X - X_*)K_{X_*,X}$$

$$\alpha = [K_{X,X} + \sigma_n^2 I]^{-1}\mathbf{y}$$

- Second-order sensitivities $\to$ diff. wrt. X $_*$

Summary on GPs

- For a fairly simple idea, Gaussian processes do tend to work very well on a wide range of topics.

- The way that the covariance function explicitly encodes the correlations that can be seen in the data means that the user has a lot of control.

- Even in the simple treatment here we have put quite a lot of effort into making the computations numerically stable and relatively fast.

- However, there is much more that can be done, including methods for approximation to speed things up significantly.